Ultimate Automizer with Two-track Proofs *
(Competition Contribution)

Matthias Heizmann[1], Daniel Dietsch[1], Marius Greitschus[1],
Jan Leike[2], Betim Musa[1], Claus Schätzle[1], and Andreas Podelski[1]

[1] University of Freiburg, Germany
[2] The Australian National University

Abstract. Ultimate Automizer is a software verification tool that implements an automata-based approach for the analysis of safety and liveness problems. The version that participates in this year's competition is able to analyze non-reachability, memory safety, termination, and overflow problems. In this paper we present the new features of our tool as well as the instructions how to install and use it.

1 Verification Approach

Ultimate Automizer implements an automata-based approach to software verification that we call *trace abstraction*[4]. The key concept in this approach is the notion of a *trace* which is a sequence of program statements. We consider a program as a set of traces, namely the set of all traces that are labellings of paths in the control flow graph. For the verification of a property, we start with all traces that potentially violate the property, e.g., for checking non-reachability of an error location we start with all traces that lead from the initial location to the error location. Then, we iteratively prove that all these traces are infeasible, i.e., we prove that none of these traces corresponds to a concrete program execution. In each iteration we take a sample trace π that potentially violates the property and analyze its feasibility. If the trace π is feasible, we found a concrete counterexample to the validity of the property. Otherwise, we construct a proof for the infeasibility of π. Next, we generalize the trace π to a set of traces that are infeasible and whose infeasibility can be shown using the proof that was constructed for π.

We use automata to represent sets of traces. The underlying alphabet is the set of all program statements. The traces that potentially violate the non-reachability property are the words that are accepted by the automaton that resembles the control flow graph of the program and whose final state is the node that corresponds to the error location of the program. The procedure for obtaining sample traces is implemented as an emptiness check and in each iteration we use a difference operation on automata to ensure that we exclude all traces whose infeasibility was already shown.

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR14 AVACS)

In the following we present new features of this year's competition candidate.

Two-track proofs. In former versions of our tool, the above mentioned infeasibility proof for a trace was an inductive sequence of state predicates. Such a sequence was obtained via Craig interpolation or via a technique that combines unsatisfiable cores, live variables and the post predicate transformer. In this year's competition contribution, we use this technique to compute two sequences of predicates. One sequence is obtained by the post predicate transformer, the other sequence is obtained by the wp predicates transformer. A second sequence of predicate is redundant to prove the infeasibility of the trace π but it improves the generalization from one infeasible trace π to a set of infeasible traces.

Semi-deterministic Büchi automata. In our termination analysis we consider infinite traces and use Büchi automata to represent sets of traces[5]. The subtraction of traces whose infeasibility was already proven involves the complementation of Büchi automata which is known to be expensive. In order to overcome this bottleneck, we adjusted our algorithm such that the input of complementation operations is always a semi-deterministic Büchi automaton. This allows us to use a specialized complementation whose result has at most 4^n states[2].

Bitprecise analysis. We use SMT-LIB to represent sets of program states and the transition relation of program statements. First, we try to verify a program by using the theory of (mathematical) integers. In order to soundly capture the semantics of machine integers we use modulo operations and we overapproximate bitwise operations, e.g., bitshifts, by a havoc operation. Whenever this analysis returns a counterexample that contains an overapproximated bitwise operation, we redo the analysis and use the SMT-LIB theory of bitvectors.

2 Software Project

Ultimate Automizer is one toolchain of the Ultimate program analysis framework. Our competition candidate uses several libraries provided by Ultimate, e.g., an automata library, the LassoRanker library which is used for the termination analysis of lasso-shaped infinite traces [6], the SMT solver SMTInterpol [3], and an interface that allows us to communicate with any SMT-LIBv2 compatible SMT solver, The source code is available on Github[3] and several toolchains of Ultimate are available via a web interface.

3 Tool Setup and Configuration

A zip archive that contains the competition candidate is available at the website of Ultimate Automizer[4]. The archive contains a binary of Z3[5] and the installation of external tools is not required. Furthermore, the archive contains the

[3] https://github.com/ultimate-pa

[4] https://ultimate.informatik.uni-freiburg.de/automizer/

[5] https://github.com/Z3Prover

Python script Ultimate.py, which maps the input given in the competition to the arguments that are required by the actual binary of Ultimate. At the SV-COMP the input to a tool is a C program inputfile, a property file prop.prp, an architecture which is either 32bit or 64bit, and a memory model which is either simple or precise. Given these arguments, the script should be invoked by the following command.

```
./Ultimate.py prop.prp inputfile 32bit|64bit simple|precise
```

The output of Ultimate Automizer is written to the file Ultimate.log and the result is written to stdout. When using BenchExec the output can be translated by the ultimateautomizer.py tool-info module[6].

If the checked property does not hold, a human readable counterexample is written to UltimateCounterExample.errorpath and an error witness is written to witness.graphml.

4 Witness Validator

Verifiers that participate in the SV-COMP output an *error witness* [1] if they find a violation of the given property. An error witness is a machine readable counterexample to the validity of the property. An error witness may not represent a single program execution that violates the property, it may represent a set of program executions. The idea is that it narrows down the space in which verifiers have to search for possible violations of the property.

Ultimate Automizer can be used to validate error witnesses. For validating an error witness wtns.graphml we invoke the command mentioned in the preceding section and append wtns.graphml as a fifth argument.

```
./Ultimate.py prop.prp inputfile 32bit|64bit simple|precise wtns.graphml
```

The witness is confirmed if and only if Ultimate Automizer reports a violation of the property. I.e., the witness is confirmed if and only if a counterexample was found in the search space restricted by the witness.

References

1. D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *ESEC/FSE*, pages 721–733. ACM, 2015.
2. F. Blahoudek, M. Heizmann, S. Schewe, J. Strejcek, and M.-H. Tsai. Complementing semi-deterministic Büchi automata. In *TACAS*, 2016.
3. J. Christ and J. Hoenicke. Cutting the mix. In *CAV 2015*, pages 37–52, 2015.
4. M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *CAV*, pages 36–52, 2013.
5. M. Heizmann, J. Hoenicke, and A. Podelski. Termination analysis by learning terminating programs. In *CAV*, pages 797–813, 2014.
6. J. Leike and M. Heizmann. Ranking templates for linear loops. *Logical Methods in Computer Science*, 11(1), 2015.

[6] http://sv-comp.sosy-lab.org/2016/systems.php